Visual Programming and Program Visualization – Towards an Ideal Visual Software Engineering System –

Sassi Bentrad[1] and Djamel Meslati[2]
[1] University Badji Mokhtar-Annaba /Computer Science, Annaba, Algeria
Email: sassi_bentrad@hotmail.fr
[2] University Badji Mokhtar-Annaba /Computer Science, Annaba, Algeria
Email: meslati_djamel@yahoo.com

***Abstract*— There has been a great interest recently in systems that use graphics to aid in the programming, debugging, and understanding of computer systems. The ''Visual Programming'' and ''Program Visualization'' are exciting areas of active computer science research that show promise for improving the programming process, for this they have been applied to these systems. This article attempts to provide more meaning to these terms by giving precise definitions, and then surveys a number of systems that can be classified as providing Visual Programming or Program Visualization. These systems are organized by classifying them into two different taxonomies. The paper also gives a brief description of our approach that concentrated on both Visual Programming and Program Visualization for an Ideal Visual Software Engineering System. We consider it as a new promising trend in software engineering.**

***Index Terms*— visual language, visual programming, program visualization.**

I. Introduction

Today industrial systems require more complex software development with high qualities in terms of maintainability and reusability. The development of software is a complicated and tedious process, requiring highly specialized skills in systems programming. It is well-known that conventional programming languages are difficult to learn and use, requiring skills that many people do not have [1]. However, there are significant advantages to supplying programming capabilities in the user interfaces of a wide variety of programs. Recently, however, a software framework has been introduced to reduce the development time and costs associated with programming. For example, the success of spreadsheets can be partially attributed to the ability of users to write programs (as collections of ''formulas''). As the distribution of personal computers grows, the majority of computer users now do not know how to program, however. They buy computers with packaged software and are not able to modify the software even to make small changes. In order to allow the end user to reconfigure and modify the system, the software may provide various options, but these often make the system more complex and still may not address the users' problems. ''Easy-to-use'' software, such as ''Direct Manipulation'' systems [2] actually make the user—programmer gap worse since more people will be able to use the software (since it is easy to use), but the internal program code is now much more complicated (due to the extra code to handle the user interface). Therefore, we must find ways to make the programming task more accessible to users. One approach to this problem is to investigate the use of graphics as the programming language. This has been called ''Visual Programming'' or ''Graphical Programming''. Some Visual Programming systems have successfully demonstrated that non-programmers can create fairly complex programs with little training [3]. Another class of systems try to make programs more understandable by using graphics to illustrate the programs after they have been created. These are called ''Program Visualization'' systems and are usually used during debugging or when teaching students how to program [4] [5]. The rest of this paper is organized as follows. The next section attempts to provide a more formal definition of these terms. The third section discusses why graphical techniques are appropriate for use with programming. Then, the various approaches to Visual Programming and Program Visualization are illustrated through a survey of relevant systems. This survey is organized around two taxonomies presented in Section 4. In Section 5, we give an overview of an evaluation of visual programming and program visualization. Section 6 discuss about Visual Programming versus Program Visualization. In Section 7, some general problems and areas for further research are addressed. Finally Sections 8 present our brief conclusion.

II. Definitions

- **Programming.** In this paper, a computer ''program'' is defined as ''a set of statements that can be submitted as a unit to some computer system and used to direct the behavior of that system" [6]. While the ability to compute ''everything'' is not required, the system must include the ability to handle variables, conditionals and iteration, at least implicitly.
- **Interpretive vs. Compiled.** Any programming language system may either be ''interpretive'' or ''compiled.'' A compiled system has a large processing delay before statements can be run while they are converted into a lower-level representation in a batch fashion. An interpretive system allows statements to be executed when they are entered. This characterization is actu-ally more of a continuum rather than a dichotomy since even interpretive languages like Lisp typically require groups of statements (such as an entire

DOI: 01.IJIT.01.03.22

procedure) to be specified before they are executed.

• **Visual Programming.** ''Visual Programming'' (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Although this is a very broad definition, conventional textual languages are not considered two dimensional since the com-pilers or interpreters process them as long, one-dimensional streams. Visual Programming does not include systems that use conventional (linear) programming languages to define pic-tures, such as, Sketchpad, CORE, PHIGS, the Macintosh Toolbox, or X-II Window Manager Toolkit [7]. It also does not include drawing packages like Apple Macintosh MacDraw, since these do not create ''programs'' as defined above. Visual programming contains various graphical approaches to specify programs.

• **Program Visualization.** ''Program Visualization'' (PV) is an entirely different concept from Visual Programming. In Visual Programming, the graphics is used to create the program itself, but in Program Visualization, the program is specified in a conventional, textual manner, and the graphics and animations is used to illustrate some aspects of the program or its run-time execution. Program Visualization systems can be utilized in program development, research, and teaching to help programmers and learners understand the structure, abstract and concrete execution as well as the evolution of software. Unfortunately, in the past, many PV systems have been incorrectly labeled "Visual Programming" (as in [8]). PV systems can be classified using two axes: whether they illustrate the code, data or algorithm of the program, and whether they are dynamic or static.

''Data Visualization" systems show pictures of the actual data of the pro-gram. Similarly, *''Code Visualization"* illustrates the actual program text, by adding graphical marks to it or by converting it to a graphical form (such as a flowchart). Systems that illustrate the *''algorithm"* use graphics to show abstractly how the program operates. This is different from data and code visualization because with algorithm visualization, the pictures may not correspond directly to data in the program, and changes in the pictures might not correspond to specific pieces of the code. For example, an algorithm animation of a sort rou-tine might show the data as lines of different heights, and swaps of two items might be shown as a smooth animation of the lines moving. The ''swap'' operation may not be explicitly in the code, however.

''Dynamic" visualizations refer to systems that can show an animation of the program running, whereas *''static"* systems are limited to snapshots of the program at certain points.

• **Visual Languages.** ''Visual Languages'' refer to all systems that use graphics, including Visual Programming and Program Visualization systems. Although all these terms are somewhat similar and confusing, it is important to have different names for the different kinds of systems, and these are the names that are conventionally used in the literature.

III. Advantages of Using Graphics

Visual Programming and Program Visualization are very appealing ideas for a number of reasons. The human visual system and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in a one-dimensional textual form, not utilizing the full power of the brain. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs; have long been known to be helpful aids in program understanding [9]. Clarisse [10] claims that graphical programming uses information in a format that is closer to the user's mental representations of problems, and will allow data to be processed in a format closer to the way objects are manipulated in the real world. It seems clear that a more visual style of programming could be easier to understand and generate for humans, especially for non-programmers or novice programmers. Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (often de-emphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers. This may be especially true during debugging, where graphics can be used to present much more information about the program state (such as current variables and data structures) than is possible with purely textual displays. This is one of the goals of Program Visualization. Other Program Visualization systems use graphics to help teach computer programming. Also, some types of complex programs, such as those that use concurrent processes or deal with real-time systems, are difficult to describe with textual languages so graphical specifications may be more appropriate. The popularity of ''direct manipulation'' interfaces [2], where there are items on the computer screen that can be pointed to and operated on using a mouse, also contributes to the desire for Visual Languages. Since many Visual Languages use icons and other graphical objects, editors for these languages usually have a direct manipulation user interface. The user has the impression of more directly constructing a program rather than having to abstractly design it.

IV. Taxonomies of Visual Languages

This paper presents two classifications. The first, discussed in the subsection A, lists the various ways that Visual Programming systems have represented the program. The second classification (subsection B) is for Program Visualization systems, and shows whether the systems illustrate the code, data or algorithm of programs. Of course, a single system may have features that fit into various categories and some systems may be hard to classify, so these taxonomies attempt to characterize the systems by their most prominent features. Also, the systems discussed here are only representative; there are many systems that have not been. Since there are so many visual language systems, it would be impossible to survey them all in a single article.

A. Classification of Visual Programming Systems (VPSs)

VPSs (or VPLs: Visual Programming Languages) can be classified in different categories according to their general

DOI: 0.IJIT.01.03. 22

program representation paradigm [11]. As the field of VPSs has matured, more and more interest has been focused on creating a robust, standardized classification for work in the area. Such a classification system not only aids researchers in finding related work but also provides a baseline with which to compare and evaluate different systems. Some of the most important names in the field, including Chang, Shu, and Burnett, have worked on identifying the defining characteristics of the major categories of VPSs [17, 16, 11].The following presents a summary of the classification scheme discussed below:

- Purely visual language
- Hybrid text and visual systems
- Others, such as Programming-by-example systems, Constraint-oriented systems and Form-based systems

Note that the categories are by no means mutually exclusive. Indeed, many languages can be placed in more than one category. The single most important category has to be purely visual languages that have been derived entirely or predominantly from graphical rules. Such languages are characterized by heavy reliance on visual techniques throughout the programming process. The programmer manipulates icons or other graphical representations to create a program which is subsequently debugged and executed in the same visual environment. The program created is compiled directly from its visual representation and is never translated into an interim text-based language. Examples of these languages include Pictorial Janus, VIPR, Prograph and PICT. There are other suggestion to subdivide this category into more specific language paradigm such as object-oriented, functional, imperative and logic. Burnett and Baker [11] have developed a classification scheme for classifying VPS research papers. Their aim is to help other researchers to easily locate relevant works in this field. Another subset of VPSs favors the idea of integration of visual languages with well-established textual programming languages. They perceive the integration might be more likely to meet the actual requirements of practical software development than the highly ambitious goal of creating purely visual languages. These hybrid systems may include both programs created visually and then translated into high-level textual language, or programs which involve the use of graphical elements in a textual language. The work of Andrew, Erwig and Meyer [12] are examples of programs created graphically and the system generates textual program from it. The latter includes works to extend existing textual language such as C++ and Basic. The current commercial system in this category includes Visual Basic and Visual C++. Beside the two major categories discussed, there exist many VPSs which fall into smaller classifications similar to Pygmalion or Sketchpad. They are termed constraint-based languages which are especially popular for simulation design. An example of this is ThingLab II [13], which is an object-oriented and constrainted programming system. A few other VPSs follow the metaphor of spreadsheets and these languages are classified as form-based VPSs. Programming using form-based VPSs involves altering a group of interconnected cells and allowing the programmer to visualize the execution of a program as a sequence of different cell states which progress through time. Examples of form-based VPSs are Forms/3, ASP (Analytic Spreadsheet Package), NoPumpG, NoPumpII and Penguims. Besides that, there are VPSs such as Vampire [14], ChemTrains and BITPICT which combine visual object-oriented programming language with a rule-base approach.

TABLE I. CLASSIFICATION OF VISUAL PROGRAMMING LANGUAGES.

Purely Visual Languages Systems	Hybrid text & Visual Programming Systems	Others
VIPR, Prograph, Pictorial Janus, LabVIEW, Visual Engineering Environment (VEE), HI-VISUAL, Vista	Visual Basic, VisualWorks, Visual J++, Visual C++, OpenStep, ObjectWorld, Rehearsal World	Pygmalion, ThingLab, Forms/3, Pursuit, Vampire, ChemTrains, BITPICT, ASP, NoPumpF, NoPumpII, Penguims.

It is important to note that in each of the categories mentioned above, we can find examples of both general-purpose VPSs and languages designed for domain-specific applications. Table 1 summarizes classification of most of the visual system or VPSs frequently quoted according to the three categories highlighted above. The field of visual programming has evolved greatly over the last ten years. Continual development and refinement of languages in the categories discussed above have led to some work which was initially considered to be part of the field being reclassified as related to but not actually exemplifying visual programming. These VPS orphans, so to speak, include algorithm animation systems, such as BALSA [17], which provide interactive graphical displays of executing programs and graphical user interface development tools, like those provided with many modern compilers including Microsoft Visual C++. Both types of systems certainly include highly visual components, but they are more graphics applications and template generators than actual programming languages.

B. Classification of Program Visualization Systems (PVSs)

The systems discussed in this section are not programming systems since code is created in the conventional manner. Therefore, none of the systems discussed below appears in the previous sections. Graphics is used here to illustrate some aspect of the program after it is written. Table 2 shows some Program Visualization systems classified by whether they attempt to illustrate the code, data or algorithm of a program, and whether the displays are static or dynamic. Some systems fit into multiple categories, because they illustrate multiple aspects or have different modes.

DOI: 01.IJIT.01.03.22

TABLE I. CLASSIFICATION OF PROGRAM VISUALIZATION SYSTEMS.

	Static	Dynamic
Code	Flowcharts SEE Visual Compiler PegaSys TPM	BALSA PV Prototype MacGnome Object-Oriented Diagrams TPM
Data	TX2 Display Files Incense	Linked Lists MacGnome
Algorithm	Stills	Two Systems Sorting out Sorting BALSA Animation Kit PV Prototype ALADDIN Animation by Demonstration TANGO

V. EVALUATION OF VISUAL PROGRAMMING AND PROGRAM VISUALIZATION

Visual Languages are new paradigms for programming, and clearly the existing systems have not been completely convincing. The challenge clearly is to demonstrate that Visual Program-ming and Program Visualization can help with real-world problems. The key to this, in my opinion, is to find appropriate domains and new domains to apply these technologies to. For general-purpose programming by professional programmers, textual languages are probably more appropriate. However, we will find new domains and new forms of Visual Language where using graphics will be beneficial. The systems discussed in this paper show that some successful areas so far include:

For Visual Programming:

- Helping to teach programming (FPL, Pict, etc.),
- Allowing non-programmers to enter information in limited domains (OPAL, spread-sheets),
- Allowing non-programmers to construct animations (PLAY) and simple computerized lessons for computer-aided instruction (Rehearsal World),
- Helping with the construction of user interfaces (Peridot, State Transition UIMS), and
- Most significantly, financial planning with spreadsheets.

For Program Visualization:

- Helping to teach algorithms involving data structures (Sorting out Sorting, BALSA),
- Helping to teach program concepts, such as Prolog code execution (TPM), and
- Helping to debug programs (MacGnome).

VI. VISUAL PROGRAMMING VERSUS PROGRAM VISUALIZATION

Graphical interfaces fall within one of two categories: Visual Programming (VP) or Program Visualization (PV), both of which are subclasses of Visual Languages. The former category is comprised of interfaces which allow the user to create programs graphically, whereas the latter category is comprised of interfaces which convert previously-written conventional code to a viewable form. In the past, the domains of these two categories have not intersected; VP interfaces allow for the creation of programs from graphical elements representing specific static-code procedures, whereas PV interfaces permit graphical viewing of arbitrary (though finite in range) procedures within a program without any capability to change the program. However, in a reconfigurable system, the user must be able to manipulate pre-existing dynamic arbitrary procedures to create programs. Thus, traditional visual languages, VP and PV, are not useful for a reconfigurable system when taken separately; the system in fact requires both.

A. A Round-Trip Visual Engineering

In classical programming languages, programs are represented as text, and the meaning results from the linear order of lexical elements. Visual programs consist of graphical and often also textual elements. The meaning of the programs depends on the spatial placement of and the connection between these elements. In accordance with various programming paradigms, there are control-flow, data-flow, functional, object-oriented, rule-based, form-based, and hybrid (multi-paradigm) visual programming languages. Many visual programming languages just allow the user more or less to visually build the abstract syntax tree of a textual program[1]. If a program created using Visual Programming is to be displayed or debugged, clearly this should be done in a graphical manner, which might be considered a form of Program Visualization. However, it is more accurate to use the term Visual Programming for systems that allow the program to be created using graphics, and Program Visualization for systems that use graphics only for illustrating programs after they have been created. The distinction between the Visual Programming Language and Visualization subareas is that visualization employs visual techniques to display data, software, algorithms and programs, but does not use these techniques to program visually.

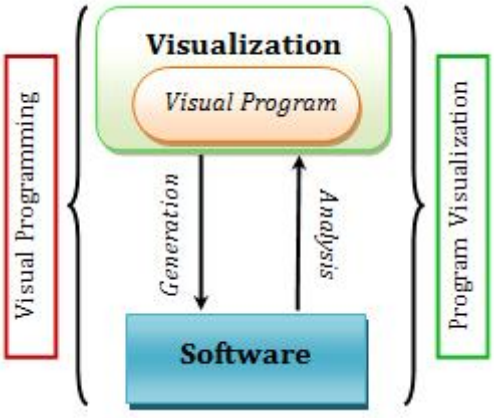

Figure 1. Visual Programming vs. Program Visualization

[1] There are also integrated development environments for textual programming languages that have been called "visual" for marketing reasons.

DOI: 0.IJIT.01.03. 22

As shown in Figure 1, visual programming and program visualization complement each other. Program visualization generates visualizations from specifications of software systems, while visual programming generates software systems from visual specifications. Combining the two approaches allows *Round-Trip Visualization*, for example by producing a visual presentation from the source code of a system, changing this visual presentation, and generating a new system. By such a definition, the system which we hope to achieve through our approach proposed would technically be referred to as a VP system, since the "only" qualifier in the PV definition eliminates any other possibility. That is, a VP system can have some PV aspects, such as graphical debugging, and yet still be considered VP. However, it is unclear in our minds that such definitions were meant to apply to a system with extensive use of both PV and VP techniques, as would be needed in a reconfig-urable system. We therefore anticipate a need for a new class of visual language which extensively incorporates both PV and VP techniques. Such a hybrid visual programming language would permit the graphical creation of pro-grams (as in VP) from visualized pre-existing conventional code (as in PV). The pre-existing conventional code could be adjusted external to the system, and subsequent system sessions would then reflect these changes using PV techniques. The code could then be configured interactively us-ing VP techniques. These ideas are included in our approach and will be implementing within a software framework which through him we hope to achieve an Ideal Visual Software Engineering System.

B. The Conceptual Model

To explain the visual programming process supported by a VPL, we use a high level conceptual model to illustrate the roles of the user, user interface, and visual program. This model also clearly defines the differences between visual programming and program visualization, which play complementary roles in software development. We adapt the model of van Wijk (2006) that was used to identify the major ingredients, costs and gains in the field of visualization. The adapted model, illustrated in Figure 2, considers the major ingredients of the *user*, *user-interface*, and *program*, rather than those of the *user*, *visualization*, and *data* in the context of visualization.

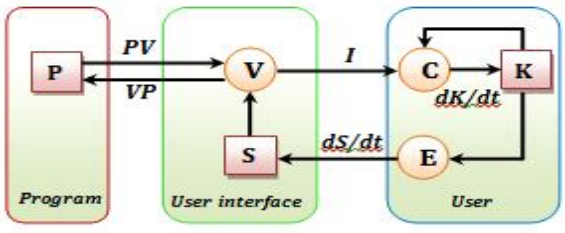

Figure 2. A Conceptuel Model of Visual Programming and Program Visualization

The boxes in Figure 2 denote containers; add circles denote processes that transform inputs into outputs. The user is modeled with his/her knowledge ***K*** about the program to be constructed, or to be understood and analyzed. The knowledge ***K*** is obtained through the user's cognitive capability ***C***, particularly the perceptual ability in the context of visual programming and program visualization. The knowledge also enhances the cognitive capability and plays the key role in driving the interactive exploration ***E*** through the user interface. Through the user interface, the user provides specifications ***S*** for the program to be developed or the algorithms and their parameters to be applied. Upon the specification supplied, a visual program ***V*** is displayed and ed-ited as an image ***I***. In the context of program visualization, ***V*** represents the visualization of a program's properties, such as status, structure, interac-tion among its components, or output results. Its image ***I*** is perceived by the user, with an increase in knowledge ***K*** as a result. Program ***P*** is what the user is interested. It is to be developed in the case of visual programming, or comprehended and analyzed in the case of pro-gram visualization. The program in this conceptual model has a broader sense than the traditionally understood program as defined by Graham (1987). ***P*** can be any of the following:

- A code sequence conforming to a traditional programming language such as Java,
- A code sequence conforming to a mark-up language such as XML which may not necessarily carry any computation, or
- A binary code or data structure generated (e.g. by a parser) from a high-level specification.

As defined in Section 2, visual programming (VP) refers to a process in which the user specifies programs in a two or more dimensional fashion, i.e. in the direction of ***V*** to ***P.*** Program visualization (PV), on the other hand, refers to a process in which certain properties of a program are displayed in a two or more dimensional fashion according to the user's selec-tion of parameters and/or algorithms. The process of PV is clearly in the direction of ***P*** to ***V***. Usually, a VPL aims at easing the process of program specification ***S*** through graphical interaction and direct manipulation with a minimal requirement of the programming knowledge. The easiness of the specification ***S*** for a given program ***P*** is measured by the amount of time ***T*** required, represented by ***dS/dt,*** While a PV system aims at maximizing the user's gain in his/her knowledge ***K*** about the program ***P*** under analysis. The measurement of PV's effectiveness is made when the user takes time ***T*** to gain additional knowledge K(T)-K(0) about the program ***P,*** represented by ***dK/dt***. An *Ideal Visual Software Engineering System* should support *Round-Trip Vis-ual Engineering* by incorporating both visual programming and program visualization. Consistent graphical formalisms in both ***VP*** and ***PV*** are desirable in order to maintain the user's mental map (Misue et al, 1995) throughout the life-cycle of the development.

VII. General Problems and Areas for Future Research

As described in the previous section, the largest area for future research is to prove that Visual Languages will actually help users. Although these systems are attractive for a number of reasons, and some have been successfully used, they share

DOI: 01.IJIT.01.03.22

a number of more technical problems unsolved which are fruitful areas for future research.

A. All Visual Languages

The problems mentioned in this section apply to many Visual Programming and Program Visualization Systems.

• **Difficulty with large programs or large data.** Almost all visual representations are physically larger than the text they replace, so there is often a problem that too little will fit on the screen. This problem is alleviated to some extent by scrolling and various abstraction mechanisms.

• **Need for automatic layout.** When the program or data gets to be large, it can be very tedious for the user to have to place each component, so the system should lay out the pic-ture automatically. Unfortunately, for many graphical representations, generating an attractive layout can be difficult, and generating a perfect layout may be intractable. For example, generating an optimal layout of graphs and trees is NP-Complete [20]. More research is needed, therefore, on fast layout algorithms for graphs that have good user interface characteristics, such as avoiding large scale changes to the display after a small edit.

• **Lack of formal specification.** Currently, there is no formal way to describe a Visual Language. Something equivalent to the BNFs used for textual languages is needed. This would provide the field with a ''hard science'' foundation, and may allow tools to be created that will make the construction of editors and compilers for Visual Languages easier. Chang [19] [21], Glinert [22] and Selker [23] have made attempts in this direc-tion, but much more work is needed.

• **Tremendous difficulty in building editors and environments.** Most Visual Languages require a specialized editor, compiler, and debugger to be created to allow the user to use the language. With textual languages, conventional, existing text editors can be used and only a compiler and possibly a debugger need to be written. Currently, each graphical language requires its own editor and environment, since there are no general purpose Visual Language editors. These editors are hard to create because there are no ''editor--compilers'' or other similar tools to help. The ''compiler-compiler'' tools used to build compilers for textual languages are also rarely useful for building compilers and inter-preters for Visual Languages. In addition, the language designer must create a system to display the pictures from the language, which usually requires low-level graphics pro-gramming. Other tools that traditionally exist for textual languages must also be created, including pretty-printers, hard-copy facilities, program checkers, indexers, cross-referencers, pattern matching and searching (e.g., ''grep'' in Unix), etc. These problems are made worse by the historical lack of portability of most graphics programs.

• **Lack of evidence of their worth.** There are not many Visual Languages that would be generally agreed are ''successful,'' and there is little in the way of formal experiments or informal experience that shows that Visual Languages are good. It would be interesting to see experimental results that demonstrated that visual programming techniques or iconic languages were better than good textual methods for performing the same tasks. Metrics might include learning time, execution speed, retention, etc. Fortunately, prelim-inary results are appearing for the advantages of using graphics for teaching students how to program [18].

• **Poor representations.** Many visual representations are simply not very good. Programs are hard to understand once created and difficult to debug and edit. This is especially true once the programs get to be a non-trivial size.

• **Lack of Portability of Programs.** A program written in a textual language can be sent through electronic mail, and used, read and edited by anybody. Graphical languages require special software to view and edit; otherwise they can only be viewed on hard-copy.

B. Specific Problems for Visual Programming

• **Lack of functionality.** Many VP systems work only in a limited domain.

• **Inefficiency:** Most VP systems run programs very slowly.

• **Unstructured programs.** Many VP systems promote unstructured programming practices in the software engineering sense (like GOTO). Many do not provide abstraction mechanisms (procedures, local variables, etc.) which are necessary for programs of a reasonable size.

• **Static representations of programs that are hard to under stand.** For flowcharts, AMBIT and similar systems, the pro-gram begins to look like a maze of wires. For Rehearsal World and similar • systems, the static representation is simply normal linear code.

• **No place for comments.** An interesting point is that virtually no VP system provides a place for comments.

• **Many Visual Programs do not integrate with programs created in different languages, such as text.** A Visual Program might be appropriate for some aspects of the programming task but not others. An exception is MPL (A Graphical Programming Environment for Matrix Processing based on Logic and Constraints) which uses a Visual Language for matrices and a textual language for everything else. Another approach is for the compiler for the visual programming language to generate conventional computer programs (e.g., in C), so they can be combined with other programs.

C. Specific Problems for Program Visualization

• **Difficulty in specifying the display.** Newer Program Visualization systems are beginning to ease the task of specifying the display, but it can still be very difficult to design and pro-gram the desired graphics. Some systems, such as BALSA-II make it easy to choose from a pre-defined set of displays, but creating other displays can still be very difficult because it involves making low-level calls to the graphics primitives.

• **Problem of controlling timing.** For dynamic data visualization, it is difficult to specify when the displays should be updated. Issues of aesthetics in timing are very important to produce useful animations.

VIII. Summary

Visual Programming and Program Visualization are exciting areas of active computer science research that show promise for improving the programming process (i.e. to improve the user interface to programming environments), especially for non-

DOI: 0.IJIT.01.03. 22

programmers or novice programmers, but more work needs to be done. A number of interesting systems have been created in each area, and there are some that cross the boundaries. This paper has attempted to classify some of these systems and present the general problems with them in hopes that this will clarify the use of the terms and provide a context for future research. The success of spreadsheets demonstrates that if we find the appropriate paradigms, graphical techniques can revolutionize the way people interact with computers. The paper also gives a brief description of our approach that concentrated on both Visual Programming and Program Visualization for an *Ideal Visual Software Engineering Environment*. This has allowed us to reduce training time and programming time on our manipulators from weeks to days, or even hours. We believe that this novel approach will open a new-direction in the programming languages development field.

REFERENCES

[1] C. Lewis and G.M. Olson. "Can Principles of Cognition Lower the Barriers to Programming?", Empirical Studies of Programmers, vol. 2, Ablex, 1987.

[2] Ben Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages", *IEEE Computer*. pp. 57–69, Aug, 1983.

[3] Daniel C. Halbert, "Programming by Example", PhD Thesis, Computer Science Division, Dept. of EE&CS, University of California, Berkeley, 1984, Also: Xerox Office Systems Division, Systems Development Department, TR OSD-T8402, December, 1984.

[4] Brad A. Myers, "Visual Programming, Programming by Example, and Program Visualization; A Taxonomy", *Proceedings SIGCHI'86: Human Factors in Computing Systems*. Boston, MA, pp. 59–66, April 13–17, 1986.

[5] Brad A. Myers, "The State of the Art in Visual Programming and Program Visualization", Carnegie Mellon University Computer Science Department Technical Report No. CMU-CS-88-114. Feb, 1988.

[6] Dictionary of Computing. Oxford: Oxford University Press, 1983.

[7] Joel McCormack and Paul Asente, "An Overview of the X Toolkit", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada*. pp 46–55, Oct, 17-19, 1988.

[8] Robert B. Grafton and Tadao Ichikawa, eds. *IEEE Computer, Special Issue on Visual Programming*. Aug, 1985.

[9] David C. Smith, "Pygmalion: A Computer Program to Model and Stimulate Creative", Thought. Basel, Stuttgart: Birkhauser, 1977.

[10] Olivier Clarisse and Shi-Kuo Chang, "VICON: A Visual Icon Manager", Visual Languages. New York: Plenum Press, pp. 151–190, 1986.

[11] Burnett, M.M., Baker, M.J.: A classification system for visual programming languages. *J. Vis. Lang. Comput*. 5(3), pp. 287–300 ,1994.

[12] M. Erwig, and B. Meyer.: Heterogeneous Visual Languages-Integrating Visual and Textual Programming. *In 11th IEEE Symp, On Visual Languages, Darmstadt*. pp. 318–325, 1995.

[13] J. H. Maloney, A. Borning and B. N. Freeman-Benson.: Constraint Technology for User-Interface Construction in ThingLab II. *In Proceedings OOPSLA 89, ACM SIGPLAN Notices*, Vol. 24, No. 10, pp. 381–388, 1989.

[14] D. W. Mcintyre.: Design and Implementation with Vampire. In M. M. Burnett, A. Goldberg, and T. Lewis, eds.: Visual Object-Oriented Programming: Concepts and Environments. Englewood Cliffs, Prentice-Hall, 1994.

[15] C. Dos Santos, "Visualisation métaphorique tridimensionnelle de l'information", Ph.D. thesis, Ecole nationale supérieure des télécommunications, PARIS-ENST, 2002.

[16] Shu, N. C. Visual Programming Languages: A Perspective and a Dimensional Analysis, pp. 11–34. Plenum Press, 1986.

[17] Chang, S. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):pp. 29–39, January 1987.

[18] Nancy Cunniff, Robert P. Taylor, and John B. Black, "Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal", *Proceedings SIGCHI'86: Human Factors in Computing Systems, Boston, MA*. pp. 175–182, April 13–17, 1986.

[19] S. K. Chang, M. Tauber, B. Yu, and J.S. Yu, "A Visual Language Compiler", *IEEE Transactions on Software Engineering*. pp. 506–525, May, 1989.

[20] D. S. Johnson, "The NP-Completeness Column: an ongoing guide", *Journal of Algorithms*, pp. 89–99, 1982.

[21] Shi-Kuo Chang, G. Tortora, Bing Yu, and A. Guercio, "Icon Purity - Toward a Formal Theory of Icons", 1987 *Workshop on Visual Languages*. August 19–21, 1987. *Linkoping, Sweden. IEEE Computer Society*, pp. 3–16.

[22] Ephraim P. Glinert and Jakob Gonczarowski, "A (Formal) Model for (Iconic) Programming Environments", *Human-Computer Interaction-Interact'87, Elsevier Science Publishers (North Holland)*. pp. 283–290, 1987.

[23[Ted Selker and Larry Koved, "Elements of Visual Language", *IEEE Workshop on Visual Languages. October 10–12, 1988. Pittsburgh,*
PA. Computer Society Order Number 876, IEEE Computer Society Press, Terminal Annex, P.0. Box 4699, Los Angeles, CA 90051. pp. 38–43, 1988.

DOI: 01.IJIT.01.03.22